安装 Typescript

编程语言类型：

动态类型语言（Dynamically Typed Language）：运行时类型检查。

静态类型语言（Statically Typed Language）：编译时类型检查。

Typescript是什么?

- Javascript的超级（支持原生javascript语法）
- 静态类型风格的类型系统
- 支持es6到esnext的语法
- 兼容各种浏览器，各种系统和各种服务器

为什么用Typescript?

1. 程序更容易理解

   函数输入输出的参数类型，一目了然，不用看函数的说明文档了。

2. 效率更高

   在不同的代码块和定义中进行跳转，接口提示。

3. 更少错误

   编译期间能发现大部分错误，杜绝一些比较常见的错误。比如undefined.xxx。

4. 非常好的包容性

   完全兼容JavaScript，第三方库可以单独编写类型文件。

Typescript 官网地址: https://www.typescriptlang.org/zh/

使用 nvm 来管理 node 版本: https://github.com/nvm-sh/nvm

安装 Typescript:

```
npm install -g typescript

```

使用 tsc 全局命令：

```
// 查看 tsc 版本
tsc -v
// 编译 ts 文件
tsc fileName.ts

```

原始数据类型

Typescript 文档地址：Basic Types

Javascript 类型分类：

原始数据类型 - primitive values：

- Boolean
- Null
- Undefined
- Number
- BigInt
- String
- Symbol

```
let isDone: boolean = false

// 接下来来到 number，注意 es6 还支持2进制和8进制，让我们来感受下

let age: number = 10
let binaryNumber: number = 0b1111

// 之后是字符串，注意es6新增的模版字符串也是没有问题的
let firstName: string = 'viking'
let message: string = `Hello, ${firstName}, age is ${age}`

// 还有就是两个奇葩兄弟两，undefined 和 null
let u: undefined = undefined
let n: null = null

// 注意 undefined 和 null 是所有类型的子类型。也就是说 undefined 类型的变量，可以赋值给 number 类型的变量：
let num: number = undefined

```

any 类型

```
let notSure: any = 4
notSure = 'maybe it is a string'
notSure = 'boolean'
// 在任意值上访问任何属性都是允许的：
notSure.myName
// 也允许调用任何方法：
```

```
notSure.getName()

```

Array 和 Tuple

Typescript 文档地址：Array 和 Tuple

```
//最简单的方法是使用「类型 + 方括号」来表示数组：
let arrOfNumbers: number[] = [1, 2, 3, 4]
//数组的项中不允许出现其他的类型：
//数组的一些方法的参数也会根据数组在定义时约定的类型进行限制：
arrOfNumbers.push(3)
arrOfNumbers.push('abc') // Argument of type 'string' is not assignable to parameter of type 'number'.ts(2345)

// 元祖的表示和数组非常类似，只不过它将类型写在了里面 这就对每一项起到了限定的作用
let user: [string, number] = ['viking', 20]
//但是当我们写少一项 就会报错 同样写多一项也会有问题
user = ['molly', 20, true] // Type '[string, number, boolean]' is not assignable to type '[string, number]'.  Source has 3 element(s) but target allows only 2.ts(2322)

```

interface 接口

Typescript 文档地址：Interface

Duck Typing 概念：

> 如果某个东西长得像鸭子，像鸭子一样游泳，像鸭子一样嘎嘎叫，那它就可以被看成是一只鸭子。

```
// 我们定义了一个接口 Person
interface Person {
  name: string;
  age: number;
}
// 接着定义了一个变量 viking，它的类型是 Person。这样，我们就约束了 viking 的形状必须和接口 Person 一致。
let viking: Person ={
  name: 'viking',
  age: 20
}
```

```ts

//有时我们希望不要完全匹配一个形状，那么可以用可选属性：
interface Person {
    name: string;
    age?: number;
}
let viking: Person = {
    name: 'Viking'
}

//接下来还有只读属性，有时候我们希望对象中的一些字段只能在创建的时候被赋值，那么可以用 readonly 定义只读属性
interface Person {
  readonly id: number;
  name: string;
  age?: number;
}
viking.id = 9527 // Cannot assign to 'id' because it is a read-only property.ts(2540)

```

函数

Typescript 文档地址：Functions

```ts
// 来到我们的第一个例子，约定输入，约定输出
function add(x: number, y: number): number {
  return x + y
}
// 可选参数
function add(x: number, y: number, z?: number): number {
  if (typeof z === 'number') {
    return x + y + z
  } else {
    return x + y
  }
}

// 函数本身的类型
const add2: (x: number, y: number, z?:number) => number = add
```

```
16
17 // interface 描述函数类型
18 const sum = (x: number, y: number) => {
19   return x + y
20 }
21 interface ISum {
22   (x: number, y: number): number
23 }
24 const sum2: ISum = sum
```

类型推论，联合类型 和 类型断言

Typescript 文档地址：类型推论 - type inference

联合类型 - union types

```
1 // 我们只需要用中竖线来分割两个
2 let numberOrString: number | string
3 // 当 TypeScript 不确定一个联合类型的变量到底是哪个类型的时候，我们只能访问此联合类型的所有类型里共有的属性或方法：
4 numberOrString.length // Property 'length' does not exist on type 'string | number'.
  Property 'length' does not exist on type 'number'.ts(2339)
5 numberOrString.toString()
6
```

类型断言 - type assertions

```
1 // 这里我们可以用 as 关键字，告诉typescript 编译器，你没法判断我的代码，但是我本人很清楚，这里我就把它看作是一个 string，你可以给他用 string 的方法。
2 function getLength(input: string | number): number {
3   const str = input as string
4   if (str.length) {
5     return str.length
6   } else {
7     const number = input as number
8     return number.toString().length
9   }
10 }
11
```

类型守卫 - type guard

```ts
// typescript 在不同的条件分支里面，智能的缩小了范围，这样我们代码出错的几率就大大的降低了。
function getLength2(input: string | number): number {
  if (typeof input === 'string') {
    return input.length
  } else {
    return input.toString().length
  }
}
```

Class 类

面向对象编程的三大特点

- 封装（Encapsulation）：将对数据的操作细节隐藏起来，只暴露对外的接口。外界调用端不需要（也不可能）知道细节，就能通过对外提供的接口来访问该对象，
- 继承（Inheritance）：子类继承父类，子类除了拥有父类的所有特性外，还有一些更具体的特性。
- 多态（Polymorphism）：由继承而产生了相关的不同的类，对同一个方法可以有不同的响应。

类 - Class

```ts
class Animal {
  name: string;
  constructor(name: string) {
    this.name = name
  }
  run() {
    return `${this.name} is running`
  }
}
const snake = new Animal('lily')

// 继承的特性
class Dog extends Animal {
  bark() {
    return `${this.name} is barking`
  }
}

const xiaobao = new Dog('xiaobao')
```

```
console.log(xiaobao.run())
console.log(xiaobao.bark())

// 这里我们重写构造函数，注意在子类的构造函数中，必须使用 super 调用父类的方法，要不就会报错。
class Cat extends Animal {
  constructor(name) {
    super(name)
    console.log(this.name)
  }
  run() {
    return 'Meow, ' + super.run()
  }
}
const maomao = new Cat('maomao')
console.log(maomao.run())

```

类成员的访问修饰符

- public 修饰的属性或方法是公有的，可以在任何地方被访问到，默认所有的属性和方法都是 public 的
- private 修饰的属性或方法是私有的，不能在声明它的类的外部访问
- protected 修饰的属性或方法是受保护的，它和 private 类似，区别是它在子类中也是允许被访问的

类与接口

类实现一个接口

```

interface Radio {
  switchRadio(trigger: boolean): void;
}
class Car implements Radio {
  switchRadio(trigger) {
    return 123
  }
}
class Cellphone implements Radio {
  switchRadio() {
  }
}

```

```
interface Battery {
  checkBatteryStatus(): void;
}

// 要实现多个接口，我们只需要中间用 逗号 隔开即可。
class Cellphone implements Radio, Battery {
  switchRadio() {
  }
  checkBatteryStatus() {

  }
}
```

枚举 Enums

枚举 Enums

```
// 数字枚举，一个数字枚举可以用 enum 这个关键词来定义，我们定义一系列的方向，然后这里面的值，枚举成员会被赋值为从 0 开始递增的数字，
enum Direction {
  Up,
  Down,
  Left,
  Right,
}
console.log(Direction.Up) // 打印：0

// 还有一个神奇的点是这个枚举还做了反向映射
console.log(Direction[0]) // 打印：UP

// 字符串枚举
enum Direction {
  Up = 'UP',
  Down = 'DOWN',
  Left = 'LEFT',
  Right = 'RIGHT',
}
const value = 'UP'
if (value === Direction.Up) {
```

```
  console.log('go up!') // 打印：go up!
}
```

泛型 Generics

泛型 Generics

泛型（Generics）是指在定义函数、接口或类的时候，不预先指定具体的类型，而在使用的时候再指定类型的一种特性。

```
function echo(arg) {
  return arg
}
const result = echo(123)
// 这时候我们发现了一个问题，我们传入了数字，但是返回了 any

function echo<T>(arg: T): T {
  return arg
}
const result = echo(123)

// 泛型也可以传入多个值
function swap<T, U>(tuple: [T, U]): [U, T] {
  return [tuple[1], tuple[0]]
}

const result = swap(['string', 123])
```

泛型第二部分 - 泛型约束

在函数内部使用泛型变量的时候，由于事先不知道它是哪种类型，所以不能随意的操作它的属性或方法

```
function echoWithArr<T>(arg: T): T {
  console.log(arg.length) // Property 'length' does not exist on type 'T'.ts(2339)
  return arg
}

// 上例中，泛型 T 不一定包含属性 length，我们可以给他传入任意类型，当然有些不包括 length 属性，那样就会报错
```

```ts

interface IWithLength {
  length: number;
}
function echoWithLength<T extends IWithLength>(arg: T): T {
  console.log(arg.length)
  return arg
}

echoWithLength('str')
const result3 = echoWithLength({length: 10})
const result4 = echoWithLength([1, 2, 3])
const result5 = echoWithLength(123) // Argument of type 'number' is not assignable to parameter of type 'IWithLength'.ts(2345)
```

泛型第三部分 - 泛型与类和接口

```ts
class Queue {
  private data = [];
  push(item) {
    return this.data.push(item)
  }
  pop() {
    return this.data.shift()
  }
}

const queue = new Queue()
queue.push(1)
queue.push('str')
console.log(queue.pop().toFixed())
console.log(queue.pop().toFixed())

//在上述代码中存在一个问题，它允许你向队列中添加任何类型的数据，当然，当数据被弹出队列时，也可以是任意类型。在上面的示例中，看起来人们可以向队列中添加string 类型的数据，但是那么在使用的过程中，就会出现我们无法捕捉到的错误，

class Queue<T> {
  private data = [];
```

```
  push(item: T) {
    return this.data.push(item)
  }
  pop(): T {
    return this.data.shift()
  }
}
const queue = new Queue<number>()

//泛型和 interface
interface KeyPair<T, U> {
  key: T;
  value: U;
}

let kp1: KeyPair<number, string> = { key: 1, value: "str"}
let kp2: KeyPair<string, number> = { key: "str", value: 123}

// 用interface描述函数类型
interface IPlus<T> {
    (a: T, b: T): T
}
function plus(a: number, b: number): number {
    return a + b
}
function connect(a: string, b: string): string {
    return a + b
}
const fun: IPlus<number> = plus
const fun2: IPlus<string> = connect
```

类型别名 和 交叉类型

类型别名 Type Aliases

类型别名，就是给类型起一个别名，让它可以更方便的被重用。

```
// type aliases: 定义类型别名
type PlusType = (x: number, y: number) => number
function sum(x: number, y: number): number {
```

```
    return x + y
}
const sum2: PlusType = sum

// 支持联合类型（或的概念）
type StrOrNumber = string | number
let result2: StrOrNumber = '123'
result2 = 123

// 字符串字面量
type Directions = 'Up' | 'Down' | 'Left' | 'Right'
let toWhere: Directions = 'Up'
```

交叉类型 Intersection Types

```
// 交叉类型（与的概念）
interface IName  {
  name: string
}
type IPerson = IName & { age: number }
let person: IPerson = { name: 'hello', age: 12}
```

声明文件

声明文件

@types 官方声明文件库 @types 搜索声明库

```
// npm install --save @types/jquery，就可以在ts里引用第三方库。
jQuery("#app")
```

calculator.js

```
function calculator(operator, numbers) {
    if (operator === 'plus') {
        return numbers[0] + numbers[1];
    }
    else if (operator === 'minus') {
        return numbers[0] - numbers[1];
    }
```

```
}
calculator.plus = function (numbers) {
    return numbers[0] + numbers[1];
};
calculator.minus = function (numbers) {
    return numbers[0] - numbers[1];
};

```

calculator.d.ts

```
type IOperator = 'plus' | 'minus'
interface ICalculator {
    (operator: IOperator, numbers: number[]) : number;
    plus: (numbers: number[]) => number;
    minus: (numbers: number[]) => number;
}
declare const calculator: ICalculator
export default calculator

```

calculatorTest.ts

```
import calculator from "./calculator";
console.log(calculator('minus', [2, 3]))
console.log(calculator.plus([1, 2]))
```

内置类型

内置类型

```
const a: Array<number> = [1,2,3]
// 大家可以看到这个类型，不同的文件中有多处定义，但是它们都是 内部定义的一部分，然后根据不同的版本或者功能合并在了一起，一个interface 或者 类多次定义会合并在一起。这些文件一般都是以 lib 开头，以 d.ts 结尾，告诉大家，我是一个内置对象类型欧
const date: Date = new Date()
const reg = /abc/
// 我们还可以使用一些 build in object，内置对象，比如 Math 与其他全局对象不同的是，Math 不是一个构造器。Math 的所有属性与方法都是静态的。
```

```

Math.pow(2,2)

// DOM 和 BOM 标准对象
// document 对象，返回的是一个 HTMLElement
let body: HTMLElement = document.body
// document 上面的query 方法，返回的是一个 nodeList 类型
let allLis = document.querySelectorAll('li')

//当然添加事件也是很重要的一部分，document 上面有 addEventListener 方法，注意这个回调函数，因为类型推断，这里面的 e 事件对象也自动获得了类型，这里是个 mouseEvent 类型，因为点击是一个鼠标事件，现在我们可以方便的使用 e 上面的方法和属性。
document.addEventListener('click', (e) => {
  e.preventDefault()
})
```

Utility Types

Typescript 还提供了一些功能性，帮助性的类型，这些类型，大家在 js 的世界是看不到的，这些类型叫做 utility types，提供一些简洁明快而且非常方便的功能。

```

// partial，它可以把传入的类型都变成可选
interface IPerson {
  name: string
  age: number
}

let viking: IPerson = { name: 'viking', age: 20 }
type IPartial = Partial<IPerson>
let viking2: IPartial = { }

// Omit，它返回的类型可以忽略传入类型的某个属性
type IOmit = Omit<IPerson, 'name'>
let viking3: IOmit = { age: 20 }

```

配置文件

配置文件的官方文档

配置示例

```jsonc
{
  "files": ["test.ts", "test2.d.ts"], // 编译源文件
  "compilerOptions": {
    "outDir": "./output", // 编译目标的路径
    "module": "ESNext", // 编译目标的模块类型
    "target":"ES5", // 编译目标js版本
    "declaration": true // 编译生成.d.ts文件
  }
```